Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 196 (2008) 95–112
www.elsevier.com/locate/entcs

Focusing the Inverse Method for LF: A Preliminary Report

Brigitte Pientka and Xi Li[1]

School of Computer Science
McGill University
Montreal, Canada

Florent Pompigne

ENS Cachan
94235 Cachan cedex, France

Abstract

In this paper, we describe a proof-theoretic foundation for bottom-up logic programming based on uniform proofs in the setting of the logical framework LF. We present a forward uniform proofs calculus which is a suitable foundation for the inverse method for LF and prove its correctness. We also present some preliminary results of an implementation for the Horn Fragment as part of the logical framework Twelf, and compare its performance with the tabled logic programming engine.

Keywords: Type theory, logical frameworks, theorem proving

1 Introduction

Logic programming is typically thought of as a backward proof search method where we start with the query and apply backchaining. We first try to find a clause head which unifies with a given query and then try to solve its subgoals. Proof-theoretically backchaining in logic programming can be elegantly explained by uniform proofs [6] which serves as a foundation for higher-order logic programming systems such as as λ-Prolog [8], Twelf [15], or Isabelle [10]. These frameworks provide a general meta-language for the specification and implementation of formal systems, and execution of these specifications is based on the operational semantics of backchaining logic programming. However, the backchaining semantics has also

[1] Email: bpientka@cs.mcgill.ca

doi:10.1016/j.entcs.2007.09.020

several disadvantages. Many straightforward specifications may not be directly executable, thus requiring more complex and sometimes less efficient implementations and performance may be severely hampered by redundant computation. The tabled logic programming engine [11,14] in Twelf addresses these concerns. It allows the logic programming interpreter to memoize subcomputations and re-use their result later, thereby eliminating infinite and redundant computation. However, a critical potential bottleneck in this system is that the memo-table may grow large and there are a large number of suspended goals. The overhead of freezing and storing a given proof search state such that it can be resumed later on is substantial.

An alternative to backchaining in logic programming is forward chaining where we start with some axioms, and then satisfy the subgoals to conclude new facts. This idea of forward chaining has been exploited in bottom-up logic programming as found for example in magic sets [16]. Forward logic programming has potentially many advantages over the more traditional backwards logic programming approaches, since suspending computation and storing a global state, as in tabled logic programming, is completely unnecessary. It provides a sound and complete proof search procedure, where only true statements are generated and only those must be stored. Forward chaining in this sense can be naturally explained by proof search based on the inverse method (see for example [3]). In this paper, we lay the foundation for exploring forward chaining in the logical framework Twelf, and present a forward uniform proof calculus together with its correctness proof. Building on this theoretical discussion, we discuss how to turn theory into a practical implementation and report on our experience with a prototype for the Horn fragment.

This paper is structured: Section 2 we introduces briefly the syntax of LF, and Section 3 gives some example specification in LF. Section 4 we present uniform calculus together with a lifted version which has meta-variables. In Section 5, we present a forward uniform proof system together with a lifted version which is a suitable basis for inverse method for LF. Section 6, we discuss implementation issues, and report on some some preliminary results for the Horn fragment and compare it to the tabled higher-order logic programming engine.

2 Background: The logical framework LF

Our main interest in this paper is in designing a forward inverse method prover for the logical framework Twelf. Twelf supports the specification of deductive systems, given via axioms and inference rules, together with the proofs about them, and has been extensively used over the past few years in several applications. The theoretical foundation for Twelf is the logical framework LF [5]. The LF language, a dependently typed lambda-calculus, can be briefly described as follows:

$$\begin{array}{ll} \text{Kinds } K & ::= \mathsf{type} \mid \Pi x{:}A.K \\ \text{Types } A & ::= a\ M_1 \ldots M_n \mid A_1 \to A_2 \mid \Pi x : A_1.A_2 \\ \text{Normal Objects } M & ::= \lambda x.M \mid R \\ \text{Neutral Objects } M & ::= x \mid c \mid R\ M \end{array}$$

We follow recent formulations which only concentrate on characterizing normal forms [9], however this is not strictly necessary. Objects provided by the logical framework LF include lambda-abstraction, application, constants and variables. To preserve canonical forms in the presence of substitution, we rely on hereditary substitutions as defined in [9]. Types classify objects, and range over type constants a which may be indexed by objects $M_1 \ldots M_n$, as well as non-dependent and dependent function types. Viewing types as propositions, LF types can be interpreted as logical propositions. Atomic type $a\ M_1 \ldots M_n$ correspond to an atomic proposition, non-dependent function type $A_1 \to A_2$ corresponds to an implication, and the dependent function type $\Pi x{:}A.B$ can be interpreted as the universal quantifier. We will use types and formulas interchangeably.

3 Example: Bounded polymorphic subtyping

As a motivating example which illustrates also many challenges we face when designing an inverse method prover for Twelf, we consider bounded subtype polymorphism (see also Ch. 26 [12]). In this system, we enrich polymorphic types such as $\forall\alpha.T$ with a subtype relation and refine the universal quantifier to carry a subtyping constraint. This example was proposed as part of the POPLmark challenge [1] to study different meta-theoretic properties about bounded subtype polymorphism. Here our focus is primarily in executing the given specification and experimenting with it. The syntax of types can be defined as follows:

$$\begin{array}{ll} \text{Types} & T ::= \text{top} \mid \alpha \mid T_1 \Rightarrow T_2 \mid \forall\alpha \leq T_1.T_2 \\ \text{Context } \Gamma & ::= \cdot \mid \Gamma, w{:}\alpha \leq T \end{array}$$

In $\forall\alpha \leq T_1.T_2$, the type variable α only binds occurrences of α in T_2. The typing context Γ keeps track of constraints such as $\alpha \leq T$. Next, we describe a subtyping algorithm using the judgment:

$$\Gamma \vdash T \leq S \qquad \text{Type } T \text{ is a subtype of } S \text{ in the context } \Gamma$$

$$\frac{}{\Gamma \vdash T \leq \text{top}}\ \text{sa-top} \qquad \frac{\alpha \leq T \in \Gamma}{\Gamma \vdash \alpha \leq T}\ \text{sa-hyp} \qquad \frac{}{\Gamma \vdash \alpha \leq \alpha}\ \text{sa-ref-tvar}$$

$$\frac{\Gamma \vdash T_1 \leq S_1 \quad \Gamma \vdash S_2 \leq T_2}{\Gamma \vdash S_1 \Rightarrow S_2 \leq T_1 \Rightarrow T_2}\ \text{sa-arr} \qquad \frac{\Gamma \vdash \alpha \leq U \quad \Gamma \vdash U \leq V}{\Gamma \vdash \alpha \leq V}\ \text{sa-tr-tvar}$$

$$\frac{\Gamma \vdash T_1 \leq S_1 \quad \Gamma, w{:}\alpha \leq T_1 \vdash S_2 \leq T_2}{\Gamma \vdash \forall \alpha \leq S_1.S_2 \leq \forall \alpha \leq T_1.T_2}\ \text{sa-all}^{\alpha,w}$$

The description is algorithmic in the sense that general rules foreflexivity and transitivity are admissible, and for each type constructor, top, $\forall$ and $\Rightarrow$ there is one rule which can be applied. However, it is worth pointing out that while the presented characterization has pleasant meta-theoretic properties, it does not eliminate all non-determinism. While the rule for transitivity is restricted to type variables on the left side of the subtyping relation, we can satisfy the left premise with four possible rules, i.e. the rule sa-top, sa-hyp, sa-ref-tvar, and sa-tr-tvar. However, only the rule sa-hyp is really fruitful. A crucial question therefore is not only how we can implement this formal system in the logical framework, but also what is the right paradigm to execute this implementation.

We begin by encoding the object-language of polymorphic types in LF using higher-order abstract syntax, i.e. type variables α in the object language will be represented as variables in the meta-language. This is standard practice.

```
tp:type.                          arr: tp -> tp -> tp.
top: tp.                          all: tp -> (tp -> tp) -> tp.
```

We define an LF type called `tp`, with the constructors `top`, `arr`, and `all`. The type for the constructor `all` takes in two arguments. The first argument stands for the bound and has type `tp`, while the second argument represents the body of the forall-expression and is represented by the function type (`tp -> tp`).

Next we consider the implementation of the subtyping relation. Since we represent variables of the object language implicitly, we cannot generically represent sa-ref and sa-tr where both these rules are applicable for all type variables. Instead of a general variable rule, we will add *rules for reflexivity and transitivity for each type variable*. Reflexivity and transitivity rules are dynamically introduced for each type variable.

We are now ready to show the encoding of these subtyping rules in LF. We first define the constant `sub` which describes the subtyping relation. Next, we represent each inference rule in the object-language as a clause consisting of nested universal quantifiers and implications. Upper-case letters denote logic variables which are implicitly bound by a Π-quantifier at the outside.

```
sub : tp -> tp -> type.
sa_top : sub S top.
sa_arr : sub S2 T2 -> sub T1 S1
          -> sub (arr S1 S2) (arr T1 T2) .
sa_all : (Πa:tp.
           (ΠU.ΠV.sub U V ->  sub a U -> sub a V) ->
           sub a T1 -> sub a a ->
           sub (S2 a) (T2 a))
         -> sub T1 S1
         -> sub (all S1 (λa.(S2 a))) (all T1 (λa.(T2 a))).
```

Using a higher-order logic programming interpretation based on backchaining, we can read the clause `sa_arr` as follows: To prove the goal `sub (arr S1 S2) (arr T1 T2)`, we must prove `sub T1 S1` and then `sub S2 T2`. Similarly we can read the clause `sa_all`: To prove `sub (all S1 (λa.(S2 a))) (all T1 (λa.(T2 a)))`, we need to prove first `sub T1 S1`, and then assuming `tr:ΠU.ΠV. sub U V -> sub a U -> sub a V`, `w:sub a T1`, and `ref:sub a a`, prove that `sub (S2 a) (T2 a)` is true where `a` is a new parameter of type `tp`.

Unfortunately, this specification cannot directly be executed using the backward logic programming engine, since the transitivity rule does not eliminate all non-determinism. Tabled logic programming memoizes previously encountered subgoals and allows us to reuse the results later on. This enables us to execute the specification for bounded subtype polymorphism. However, tabled logic programming has also a substantial overhead of storing encountered goals together with their answer substitution, and freezing and suspending computation to resume it later.

In this paper, we explore an alternative paradigm, a forward logic programming. This means we start from some axioms and then apply the given rules in a forward direction. In this example, `sub T top` is an axiom and so is for example, `sub a a` for any variable `a`. The clause `sa_arr` is then interpreted in a forward direction as follows: Given a proof for `sub T1 S1` and a proof for `sub S2 T2`, we can derive `sub (arr S1 S2) (arr T1 T2)`. We present first a theoretical foundation for forward proof search, and then outline the basic idea and challenges when implementing an inverse method search engine based on it. Finally, we conclude with a discussion of some preliminary results of our current prototype implementation. While our prototype only concentrates on the Horn fragment, it nevertheless provides some interesting preliminary results and analysis especially when compared to tabled logic programming.

4 Uniform Proofs

A standard proof-theoretic characterization for backchaining in logic programming is based on uniform proofs [6]. The essential idea in uniform proofs is to chain all invertible rules eagerly, and postpone the non-invertible rules lead to a uniform sequent calculus. In a uniform calculus, we distinguish between uniform phase, where we apply all the invertible rules, and focusing phase, where we pick and

focus on a non-invertible rule. We can characterize uniform proofs by two main judgments:

$\Gamma \Longrightarrow A$ There is a *uniform proof* for A from the assumptions in Γ

$\Gamma \gg A \Longrightarrow P$ There is a *focused proof* for the atom P focusing on the proposition A using the assumptions in Γ

Next, we present a proof system characterizing uniform proofs.

$$\frac{\Gamma \gg A \Longrightarrow P \quad A \in \Gamma}{\Gamma \Longrightarrow P}\ \mathsf{choose} \qquad \frac{}{\Gamma \gg P \Longrightarrow P}\ \mathsf{hyp}$$

$$\frac{\Gamma, c{:}A_1 \Longrightarrow A_2}{\Gamma \Longrightarrow A_1 \to A_2}\ \to \mathsf{R} \qquad \frac{\Gamma \Longrightarrow A_1 \quad \Gamma \gg A_2 \Longrightarrow P}{\Gamma \gg A_1 \to A_2 \Longrightarrow P}\ \to \mathsf{L}$$

$$\frac{\Gamma, x{:}A \Longrightarrow B}{\Gamma \Longrightarrow \Pi x{:}A.B}\ \Pi\mathsf{R} \qquad \frac{\Gamma \gg [M/x]A \Longrightarrow P \quad \Gamma \vdash M : A}{\Gamma \gg \Pi x{:}A.B \Longrightarrow P}\ \Pi\mathsf{L}$$

We note that our context Γ keeps track of dynamic assumptions which are introduced in the rule $\to$R and can be used during proof search as well as parameter assumptions which are introduced in the rule ΠR but cannot be used in proof search. Our goal is to enforce that every proposition is well-typed, so in the sequent $\Gamma \Longrightarrow A$ we have that A is a well-formed type in the context Γ, and similarly in the sequent $\Gamma \gg A \Longrightarrow P$ we have that A and P are a well-formed types in the context Γ. Moreover, we require in the rule ΠL that M has type A in the context Γ.

In practice, we typically do not guess the correct instantiation for the universally quantified variables in the ΠL rule, but introduce a meta-variable which will be instantiated with unification later. Previously[13] , we have been advocating the use of meta-variables. Meta-variables are associated with a postponed substitution σ which is applied as soon as we know what the meta-variable stands for. A formal treatment for meta-variables based on contextual modal types can be found in [13,9]. This allows us to formally distinguish between the ordinary bound variables introduced by ΠR or a λ-abstraction and meta-variables u which are subject to instantiation. An advantage of this approach is that we localize dependencies while allowing in-place updates. Moreover, we can present all meta-variables that appear in a given term in a linear order and ensure that the types and contexts of meta-variables further to the right may mention meta-variables. When a meta-variable is introduced it is created as $u[\mathsf{id}_\Gamma]$ meaning it can depend on all the bound variables occurring in Γ. During search Γ is concrete and id_Γ will be unfolded. Moreover, we can easily characterize all the meta-variables occurring in a formula or sequent. The distinction between ordinary bound variables and meta-variables provides a clean basis for describing proof search. We will therefore enrich our lambda-calculus with first-class meta-variables denoted by u.

$$\begin{array}{lll} \text{Neutral Terms } M & ::= \ldots \mid u[\sigma] \\ \text{Meta-variable context } \Delta & ::= \cdot \mid \Delta, u{::}A[\Psi] \end{array}$$

The type of a meta-variable is $A[\Psi]$ denoting an object M which has type A in the context Ψ. We briefly highlight how contextual substitution into types and objects-level terms is defined to give an intuition, but refer the interested reader to [9] for more details. We write $[\![\hat{\Psi}.M/u]\!]$ for replacing a meta-variable u with an object M. $\hat{\Psi}$ characterizes the ordinary bound variables occurring in M. This explicit listing of the bound variables occurring in M is necessary because of α-renaming issues and can be eliminated in an implementation. We only show contextual substitution into objects here.

$$\begin{array}{lll} [\![\hat{\Psi}.M/u]\!](\lambda y.N) & = \lambda y.N' & \text{if } [\![\hat{\Psi}.M/u]\!]N = N' \\ [\![\hat{\Psi}.M/u]\!](u[\sigma]) & = M' & \text{if } [\![\hat{\Psi}.M/u]\!]\sigma = \sigma' \text{ and } [\sigma'/\Psi]M = M' \\ [\![\hat{\Psi}.M/u]\!](u'[\sigma]) & = u'[\sigma'] & \text{if } u' \neq u \text{ and } [\![\hat{\Psi}.M/u]\!]\sigma = \sigma' \\ [\![\hat{\Psi}.M/u]\!](R\ N) & = (R'\ N') & \text{if } [\![\hat{\Psi}.M/u]\!]R = R' \text{ and } [\![\hat{\Psi}.M/u]\!](N) = N' \\ [\![\hat{\Psi}.M/u]\!](x) & = x \\ [\![\hat{\Psi}.M/u]\!](c) & = c \end{array}$$

We note that there are no side-conditions necessary when substituting into λ-abstraction, since the objects M we substitute for u is closed with respect to $\hat{\Psi}$. When we encounter a meta-variable $u[\sigma]$, we first apply $[\![\hat{\Psi}.M/u]\!]$ to the substitution σ yielding σ' and then replace u with M and apply the substitution σ'. Note because of α-renaming issues we must possibly rename the domain of σ'.

Simultaneous contextual substitution can be defined following similar principles. A simultaneous contextual substitution maps the meta-variables in its domain Δ' to another meta-variable context Δ which describes its range. More formally we can define simultaneous contextual substitutions as well-typed as follows:

$$\frac{}{\Delta \vdash \cdot : \cdot} \qquad \frac{\Delta \vdash \theta : \Delta' \quad \Delta; \Psi \vdash M : A}{\Delta \vdash (\theta, \hat{\Psi}.M/u) : \Delta', u{::}A[\Psi]}$$

Finally, we are in the position to give a uniform calculus which introduces meta-variables in the rule $\Pi\mathsf{L}$, and delays their instantiation to the hyp rule where we rely on higher-order unification to find the correct instantiation. Since higher-order unification is undecidable in general we restrict it to the pattern fragment.

$\Delta; \Gamma \Longrightarrow A/(\theta, \Delta')$ There is a *uniform proof* for A from the assumptions in Γ where θ is a contextual substitution which instantiates the meta-variable in Δ and has range Δ'

$\Delta; \Gamma \gg A \Longrightarrow P/(\theta, \Delta')$ There is a *focused proof* for the atom P focusing on the proposition A using the assumptions in Γ where θ is a contextual substitution which instantiates the meta-variable in Δ and has range Δ'

In the rule $\Pi\mathsf{L}$ we introduce a new meta-variable $u[\mathsf{id}_\Gamma]$ of type $A[\Gamma]$. This means we introduce a meta-variable whose instantiation can depend on all the parameters occurring in Γ. In the hypothesis rule, we rely on higher-order pattern unification to find the most general unifier θ of P' and P, s.t. $[\![\theta]\!]P' = [\![\theta]\!]P$.

$$\frac{\Delta; \Gamma \gg A \Longrightarrow P/(\theta, \Delta') \quad A \in \Gamma}{\Delta; \Gamma \Longrightarrow P/(\theta, \Delta')} \qquad \frac{\Delta; \Gamma \vdash P' \doteq P/(\theta, \Delta')}{\Delta; \Gamma \gg P' \Longrightarrow P/(\theta, \Delta')}$$

$$\frac{\Delta; \Gamma, A_1 \Longrightarrow A_2/(\theta, \Delta')}{\Delta; \Gamma \Longrightarrow A_1 \to A_2/(\theta, \Delta')} \qquad \frac{\Delta; \Gamma \Longrightarrow A_1/(\theta_1, \Delta_1) \quad \Delta_1; [\![\theta_1]\!]\Gamma \gg [\![\theta_1]\!]A_2 \Longrightarrow [\![\theta_1]\!]P/(\theta_2, \Delta_2)}{\Delta; \Gamma \gg A_1 \to A_2 \Longrightarrow P/([\![\theta_2]\!]\theta_1, \Delta_2)}$$

$$\frac{\Delta; \Gamma, x{:}A \Longrightarrow B/(\theta, \Delta')}{\Delta; \Gamma \Longrightarrow \Pi x{:}A.B\ (\theta, \Delta')} \qquad \frac{\Delta, u{::}A[\Gamma]; \Gamma \gg [u[\mathsf{id}_\Gamma]/x]B \Longrightarrow P/((\theta, \hat{\Gamma}.M/u), \Delta')}{\Delta; \Gamma \gg \Pi x{:}A.B \Longrightarrow P/(\theta, \Delta')}$$

Next, we prove that this system is sound and complete with the uniform proofs where we guess the correct instantiation for the meta-variables.

Theorem 4.1 (Soundness)

(i) *If* $\Delta; \Gamma \Longrightarrow A/(\theta, \Delta')$ *then for any grounding substitution* $\cdot \vdash \rho : \Delta'$ *we have* $\cdot; [\![\rho]\!][\![\theta]\!]\Gamma \Longrightarrow [\![\rho]\!][\![\theta]\!]A$.

(ii) *If* $\Delta; \Gamma \gg A \Longrightarrow P/(\theta, \Delta')$ *then for any grounding substitution* $\cdot \vdash \rho : \Delta'$ *we have* $\cdot; [\![\rho]\!][\![\theta]\!]\Gamma \gg [\![\rho]\!][\![\theta]\!]A \Longrightarrow [\![\rho]\!][\![\theta]\!]P$.

Proof. By structural induction on the first derivation (see also [13]). □

Theorem 4.2 (Completeness)

(i) *If* $\cdot; [\![\rho]\!]\Gamma \Longrightarrow [\![\rho]\!]A$ *for a modal substitution* ρ, *s.t.* $\cdot \vdash \rho : \Delta$
then $\Delta; \Gamma \Longrightarrow A/(\Delta', \theta)$ *for some* θ *and* $\rho = [\![\rho']\!]\theta$ *for some* ρ' *s.t.* $\cdot \vdash \rho' : \Delta'$.

(ii) *If* $\cdot; [\![\rho]\!]\Gamma \gg [\![\rho]\!]A \Longrightarrow [\![\rho]\!]P$ *for a modal substitution* ρ *s.t.* $\cdot \vdash \rho : \Delta$
then $\Delta; \Gamma \gg A \Longrightarrow P/(\Delta', \theta)$ *for some* θ *and* $\rho = [\![\rho']\!]\theta$ *for some* ρ', *s.t.* $\cdot \vdash \rho' : \Delta'$.

Proof. Simultaneous structural induction on the first derivation (see also [13]). □

5 Inverse method and focusing

An interesting alternative to backward proof search, is forward proof search based on the inverse method. This has potentially many advantages. While backward logic programming based depth first search is incomplete and requires backtracking, forward search provides a complete search strategy without backtracking. Similar to tabling, it also allows us to execute some specifications which were not previously executable. Next, we present a forward uniform proof system where we guess the correct instantiation. We derive this forward calculus from the uniform proof system presented earlier which models backchaining. Hence our system will only distinguish between the left focusing and right uniform phase. To obtain a more general proof-theoretic foundation, one could distinguish between a right-focusing and a left-focusing phase (see for example [3]). Finally, we describe a lifted version with meta-variables.

$\Gamma \stackrel{f}{\Longrightarrow} A$ A has *forward uniform proof* using the assumptions in Γ

$\Gamma \gg A \stackrel{f}{\Longrightarrow} P$ P has a *forward focused proof* focusing on the proposition A using the assumptions in Γ

The context Γ is now interpreted differently, in that sequents $\Gamma \stackrel{f}{\Longrightarrow} A$ and $\Gamma \gg A \stackrel{f}{\Longrightarrow} P$ assert that all assumptions in Γ as well as A, if the sequent is focused are needed to prove the conclusion. General weakening is thus disallowed but incorporated in the rule $\mathsf{f}{\rightarrow}\mathsf{R}_2$. Since our context Γ keeps track of dynamic assumption and parameters, we do not require it to be completely empty in the rule f-ax. Instead we can think of it as the strongest context in which P is well-typed. Since we want to preserve that contexts Γ are well-typed, we must make sure in the rule $\mathsf{f}{\rightarrow}\mathsf{L}$ that the union two context Γ_1 and Γ_2 is well-typed and preserves the present dependencies between parameter assumptions and dynamic assumptions. The rule f-drop was called in the backwards uniform calculus choose. In the forward direction there is no choice but rather we must drop the formula out of the focus.

$$\frac{\Gamma \gg A \stackrel{f}{\Longrightarrow} P}{\Gamma \cup \{A\} \stackrel{f}{\Longrightarrow} P}\ \text{f-drop} \qquad \frac{}{\Gamma \gg P \stackrel{f}{\Longrightarrow} P}\ \text{f-ax}$$

$$\frac{\Gamma, c{:}A_1 \stackrel{f}{\Longrightarrow} A_2}{\Gamma \stackrel{f}{\Longrightarrow} A_1 \rightarrow A_2}\ \mathsf{f}{\rightarrow}\mathsf{R}_1 \qquad \frac{\Gamma \stackrel{f}{\Longrightarrow} A_2}{\Gamma \stackrel{f}{\Longrightarrow} A_1 \rightarrow A_2}\ \mathsf{f}{\rightarrow}\mathsf{R}_2 \qquad \frac{\Gamma_1 \stackrel{f}{\Longrightarrow} A_1 \quad \Gamma_2 \gg A_2 \stackrel{f}{\Longrightarrow} P}{\Gamma_1 \cup \Gamma_2 \gg A_1 \rightarrow A_2 \stackrel{f}{\Longrightarrow} P}\ \mathsf{f}{\rightarrow}\mathsf{L}$$

$$\frac{\Gamma, x{:}A \stackrel{f}{\Longrightarrow} B}{\Gamma \stackrel{f}{\Longrightarrow} \Pi x{:}A.B}\ \mathsf{f}\Pi\mathsf{R} \qquad \frac{\Gamma \gg [M/x]B \stackrel{f}{\Longrightarrow} P \quad \Gamma \vdash M : A}{\Gamma \gg \Pi x{:}A.B \stackrel{f}{\Longrightarrow} P}\ \mathsf{f}\Pi\mathsf{L}$$

Next, we prove soundness and completeness of this forward uniform calculus.

Theorem 5.1 (Soundness)

(i) *If* $\Gamma \stackrel{f}{\Longrightarrow} A$ *then* $\Gamma \Longrightarrow A$

(ii) *If* $\Gamma \gg A \stackrel{f}{\Longrightarrow} P$ *then* $\Gamma \gg A \Longrightarrow P$.

Proof. Straightforward structural induction. □

Theorem 5.2 (Completeness)

(i) *If* $\Gamma \Longrightarrow A$ *then* $\Gamma' \stackrel{f}{\Longrightarrow} A$ *where* $\Gamma' \subseteq \Gamma$.

(ii) *If* $\Gamma \gg A \Longrightarrow P$ *then* $\Gamma' \gg A \stackrel{f}{\Longrightarrow} P$ *where* $\Gamma' \subseteq \Gamma$

Proof. Straightforward structural induction. □

The forward uniform proof system presented gives rise to a proof search method based on the inverse method central to which is the notion of subformula. We outline the notion of subformulas and present a lifted calculus following the development set out in [4]. We adapt the standard definition of subformulas to the higher-order setting where objects may contain meta-variables. The immediate free subformula of the negative occurrence of the formula $\Pi x{:}A.B$ in the context Γ is then $[u[\mathsf{id}_\Gamma]/x]B$. The immediate ground subformula of the negative occurrence of the formula $\Pi x{:}A.B$ in the context Γ is $[M/x]B$. Free signed subformulas and its immediate signed subformulas are defined inductively as follows:

signed subformula	free signed subformula	immediate signed subformula
$(A \to B)^-$	A^+, B^-	A^+, B^-
$(A \to B)^+$	A^-, B^+	A^-, B^+
$(\Pi x{:}A.B)^-$	$([u[\mathsf{id}_\Gamma]/x]B)^-$	$([M/x]B)^-$
$(\Pi x{:}A.B)^+$	$([a/x]B)^+$	$([a/x]B))^+$

Definition 5.3 [Subformula property]

(i) Every derivation of a uniform sequent $\Gamma \Longrightarrow A$ consists of signed ground subformulas of signed formulas in Γ^- and A^+.

(ii) Every derivation of a focused $\Gamma \gg A \Longrightarrow P$ consists of signed ground subformulas of signed forumlas in Γ^- and A^+.

Theorem 5.4 (Ground subformula property of uniform proofs)

(i) *Let* $\mathcal{D}$ *be a derivation of a signed uniform sequent* $\Gamma^- \Longrightarrow A^+$ *then every signed uniform sequent* $\Gamma_0^- \Longrightarrow A_0^+$ *or signed focused sequent* $\Gamma_1^- \gg A_1^- \Longrightarrow P_1$ *occurring in* $\mathcal{D}$ *fulfills the subformula property, i.e.* $[\Gamma_0^-, A_0^+] < [\Gamma^-, A^+]$ *or* $[\Gamma_1^-, A_1^-, P_1^+] < [\Gamma^-, A^+]$.

(ii) *Let* $\mathcal{D}$ *be a derivation of a signed focused sequent* $\Gamma^- \gg A^- \Longrightarrow P^+$ *then every signed uniform sequent* $\Gamma_0^- \Longrightarrow A_0^+$ *or signed focused sequent* $\Gamma_1^- \gg A_1^- \Longrightarrow P_1$

occurring in $\mathcal{D}$ fulfills the subformula property, i.e. $[\Gamma_0^-, A_0^+] < [\Gamma^-, A^-, P^+]$ *or* $[\Gamma_1^-, A_1^-, P_1^+] < [\Gamma^-, A^-, P^+]$.

Proof. By routine inspection of the inference rules for uniform and focused proofs.□

Thus when we search for a proof of a particular signed sequent $\Gamma \Longrightarrow A$ or $\Gamma \gg A \Longrightarrow P$ resp. we can restrict our search to sequents consisting of signed subformulas of $[\Gamma^-, A^+]$. When $[\Gamma^-, A^+]$ contains quantifiers, it may have an infinite number of signed subformulas, so the subformula property does not restrict the search space good enough. However any signed formula has only a finite number of free signed subformulas.

Next, we consider *free signed subformula property.* We will often represent signed subformulas of a given uniform sequent $\Gamma^- \Longrightarrow A^+$ in the form $[\![\theta]\!]\Gamma_0^- \Longrightarrow [\![\theta]\!]A_0^+$, where θ is a substitution from the meta-variables Δ to some ground instance, i.e. $\cdot \vdash \theta : \Delta$ and $\Delta; \Gamma_0^- \Longrightarrow A_0^-$. We call this the representation *via free signed subformula*. Similarly we often represent signed subformulas of a given focused sequent $\Gamma^- \gg A^- \Longrightarrow P^+$ in the form $[\![\theta]\!]\Gamma_0^- \gg [\![\theta]\!]A^- \Longrightarrow [\![\theta]\!]P^+$. Moreover, we often write $S = [\Gamma^-, A+]$ as an abbreviation for the sequent $\Gamma \Longrightarrow A$, and $\Delta \vdash S$ as an abbreviation for $\Delta; \Gamma \Longrightarrow A$.

Lemma 5.5 *Let* $S_0 = [\Gamma_0^-, A_0^+]$, *and* $S_1 = [\Gamma_1^-, A_1^+]$ *be free signed subformulas s.t.* $\Delta_0 \vdash S_0$ *and* $\Delta_1 \vdash S_1$. *Then* $[\Gamma_1^-, A_1^+] < [\Gamma_0^-, A_0^+]$ *i.e.* S_1 *is a signed subformula of* S_0, *iff* $S_1 = [\![\theta]\!]S$ *for some signed sequent* S *s.t.* S *is a free signed subformula of* S_0, *where* $\Delta_1 \vdash \theta : \Delta$ *and* $\Delta \vdash S$ *and* $\Delta \cap \Delta_0 = \emptyset$.

Proof. By inspection of the definition of signed subformulas. □

Every signed subformula of a closed signed formula can be obtained from a free signed subformula by applying a contextual substitution. We now reformulate the subformula property.

Corollary 5.6 (Free subformula property) *Let* $\mathcal{D}$ *be a derivation of a closed signed uniform sequent* $S = \Gamma^- \Longrightarrow A^+$ *or closed signed focused sequent* $\Gamma^- \gg A^- \Longrightarrow P^+$. *Every signed sequent occurring in* $\mathcal{D}$ *has the form* $[\![\theta]\!]S_0$ *for a free signed sequent* S_0 *of* S *and a substitution* θ *s.t.* $\Delta \vdash S_0$ *and* $\cdot \vdash \theta : \Delta$.

Suppose we want to check the provability of a closed signed sequent S. By the previous corollary, we can restrict signed formulas occurring in the derivation to signed sequents of the form $[\![\theta]\!]S_0$ where S_0 is a free signed sequent of S s.t. $\Delta \vdash S_0$ and $\cdot \vdash \theta : \Delta$. Since this applies to axioms as well, every axiom has the form $\Gamma \gg [\![\theta]\!]([\![\rho]\!]P) \to [\![\theta]\!]P'$ where P and P' are atomic free signed subformulas of the sequent S and $[\![\theta]\!]([\![\rho]\!]P) = [\![\theta]\!]P'$, and ρ is a renaming of meta-variables occurring in P, and Γ characterizes the parameters occurring in $[\![\theta]\!]P'$ and $[\![\theta]\!]([\![\rho]\!]P)$ respectively. For any given P, P' there may be an infinite number of such axioms because of different choices for substitutions θ, but there is only a finite number of pairs of free signed sequents. We can choose a most general axiom that represents all axioms.

We will now introduce a forward calculus $\mathcal{F}^A$ for the inverse method with meta-variables. The calculus is based on the idea of representing sequents through free

subformulas and using most general unifiers. Since higher-order unification is only decidable for patterns, we restrict our attention for now to this fragment. A sequent S in the original forward calculus for closed sequents, is an instance of a sequent $[\![\theta]\!]S_0$ in the calculus $\mathcal{F}^A$ if there exists a grounding substitution ρ s.t. $[\![\rho]\!][\![\theta]\!]S_0 = S$. Unlike more standard presentation where we associate a substitution θ with each of the formulas in Γ and the conclusion A, we will associate a substitution θ with a sequent. The judgment $(\Gamma \to A) \cdot \theta$ denotes a sequent where $[\![\theta]\!]\Gamma \to [\![\theta]\!]A$. This will be easier to implement, and models more closely our prototype.

$$\frac{(\Delta;\Gamma \gg A \xrightarrow{f} P)\cdot\theta}{(\Delta;\Gamma \cup \{A\} \xrightarrow{f} P)\cdot\theta} \qquad \frac{\Delta;\Gamma \vdash [\![\rho]\!]P' \doteq P/\theta}{(\Delta;\Gamma \gg [\![\rho]\!]P' \xrightarrow{f} P)\cdot\theta}$$

$$\frac{(\Delta;\Gamma \xrightarrow{f} B)\cdot\theta}{(\Delta;\Gamma \xrightarrow{f} (A \to B))\cdot\theta} \qquad \frac{(\Delta;\Gamma, c{:}A_1 \xrightarrow{f} A_2)\cdot\theta}{(\Delta;\Gamma \xrightarrow{f} (A_1 \to A_2))\cdot\theta}$$

$$\frac{(\Delta_1;\Gamma_1 \xrightarrow{f} A_1)\cdot\theta_1 \qquad (\Delta_2;\Gamma_2 \gg A_2 \xrightarrow{f} P)\cdot\theta_2}{(((\Delta_1 \cup \Delta_2);\Gamma_1 \cup \Gamma_2) \gg (A_1 \to A_2) \xrightarrow{f} P)\cdot[\![\theta]\!]\theta_1'} \quad \begin{array}{r}\text{where } \mathrm{ext}(\Delta_1 \cup \Delta_2, \theta_1) = \theta_1' \\ \mathrm{ext}(\Delta_1 \cup \Delta_2, \theta_2) = \theta_2' \\ \mathsf{mgu}(\theta_1', \theta_2') = \theta\end{array}$$

$$\frac{(\Delta;\Gamma, x{:}A \xrightarrow{f} B)\cdot\theta}{(\Delta;\Gamma \xrightarrow{f} (\Pi x{:}A.B))\cdot\theta} \qquad \frac{(\Delta, u{::}A[\Gamma];\Gamma \gg [u[\mathsf{id}_\Gamma]/x]B \xrightarrow{f} P)\cdot(\theta, \hat{\Gamma}.M/u) \quad u \text{ is new}}{(\Delta;\Gamma \gg (\Pi x{:}A.B) \xrightarrow{f} P)\cdot\theta}$$

In the hypothesis rule where we unify the assumption $[\![\rho]\!]P'$ with P we keep a context Γ which describes the parameters occurring in P' and P. As mentioned earlier, typical formulations of forward calculi require the context to be empty, since they do not keep track explicitly of the parameters introduced during proof search. Due to the dependent nature of our calculus, and the fact that we would like to preserve that all propositions are well-typed, we keep track of parameters explicitly and allow the context Γ in this hypothesis rule to be non-empty. Our intention is that Γ describes all the parameters occurring in P and P'. This is largely straightforward. In the implication left rule, we must union not only the assumptions in Γ_1 and in Γ_2, but we also must union the meta-variables occurring in both branches. Since meta-variables occurring in both branches of the proof, may have been instantiated differently, we must reconcile their different instantiations in θ_1 and θ_2 by unifying them. Before we can unify them we first extend them with identity substitution s.t. they share the same domain. This extension is denoted with $\mathrm{ext}(\Delta_1 \cup \Delta_2, \theta_1) = \theta_1'$ and $\mathrm{ext}(\Delta_1 \cup \Delta_2, \theta_2) = \theta_2'$ respectively.

Theorem 5.7 (Soundness)

(i) *If* $(\Gamma \xrightarrow{f} A)\cdot\theta$ *then for any grounding substitution* ρ*, we have* $[\![\rho]\!]([\![\theta]\!]\Gamma) \xrightarrow{f} [\![\rho]\!][\![\theta]\!]A$.

(ii) *If* $(\Gamma \gg A \xrightarrow{f} P)\cdot\theta$ *then for any grounding substitution* ρ*,*

we have $[\![\rho]\!]([\![\theta]\!]\Gamma) \gg [\![\rho]\!]([\![\theta]\!]A) \stackrel{f}{\Longrightarrow} [\![\rho]\!][\![\theta]\!]P$.

Proof. Proof by induction on the first derivation. □

Theorem 5.8 (Completeness) *Suppose* $\Gamma \stackrel{f}{\Longrightarrow} [\![\theta]\!]A$ *(resp.* $\Gamma \gg [\![\theta]\!]A \stackrel{f}{\Longrightarrow} [\![\theta]\!]P$*) and* $\Gamma = [\![\theta_1]\!]A_1, \ldots [\![\theta_n]\!]A_n$ *where* $A^+, A_1^-, \ldots A_n^-$ *are signed free subformulas of the goal. Then there exist a substitution* θ' *and a grounding substitution* ρ *such that:*

(i) $(A_1, \ldots A_n \stackrel{f}{\Longrightarrow} A) \cdot \theta'$ *(resp.* $(A_1, \ldots A_n \gg A \stackrel{f}{\Longrightarrow} P) \cdot \theta'$*)*

(ii) $[\![\rho]\!][\![\theta']\!](A_i) = \theta_i(A_i)$ *and* $[\![\rho]\!][\![\theta']\!](A) = [\![\theta]\!](A)$ *(resp. and* $[\![\rho]\!][\![\theta']\!](P) = [\![\theta]\!](P)$*)*

Proof. Proof by induction on the first derivation. □

6 Implementation of an inverse method prover for LF

In this section, we discuss the implementation of an inverse method prover for LF by considering the example given earlier. The first step in the inverse method is computation of subformulas. Given a signed formula we compute the set $\mathcal{N}$ of negative subformulas and the set $\mathcal{P}$ of positive subformulas. Each subformula is denoted as $\Delta; \Gamma \vdash a\ M_1 \ldots M_n$ where Δ characterizes the meta-variables, and Γ describes the parameters occurring in $a\ M_1 \ldots M_n$ Given the set $\mathcal{N}$ of negative subformulas and the set $\mathcal{P}$ of positive subformulas, we can generate a *focused axioms*, if a negative subformula unifies with a positive subformula. We compute the minimal set $\mathcal{F}$ of *focused axioms* by checking forward and backward subsumption of newly generated axiom. Following Chaudhuri *et al.*, our implementation creates big-step derived rules by chaining all the focused rules together to form a focused thread and chaining all the uniform rules together to form a uniform thread. Our compiled rules are therefore of the following form

$$\frac{\stackrel{f}{\Longrightarrow} P_1 \quad \ldots \quad \stackrel{f}{\Longrightarrow} P_n}{\stackrel{f}{\Longrightarrow} P}$$

After this pre-compilation phase is finished, we delete the focused axiom, and search over the set $\mathcal{F}$ of uniform facts and the set $\mathcal{R}$ of pre-compiled derived rules.

6.1 Top-level of the inverse method

Next, we must iterate over the set $\mathcal{F}$ of uniform facts and the set $\mathcal{R}$ of pre-compiled derived rules to generate new facts by forward chaining. Essentially we need to plug the facts into the open premises to generate new facts. There are essentially two possible loop structures which we both briefly discuss. Both of these two loops have been implemented and tested for the Horn fragment.

Iteration over facts The first loop follows essentially ideas used by K. Chaudhuri in his implementation of the inverse method for linear logic [2]. We pick a fact f from the set $\mathcal{F}$ of fact and then use this fact f to generate new pre-instantiated

rules and new facts from the set $\mathcal{R}$. Given a rule with the premises $P_1, \ldots, P_n$, we try to unify each P_i with the fact f and generate all pre-instantiated rules for this given fact f. If the fact f unifies with k premises, then we generate possibly up to $2^k - 1$ pre-instantiated rules where k is less than n. If k is equal to n, i.e. all premises can be satisfied, a new fact P is generated which is added to the set $\mathcal{F}$ if there is no fact f' in $\mathcal{F}$ s.t. P is an instance of f'. The set of rules therefore may grow exponentially during execution. However, an advantage is that every fact f will be chosen only once, and only once we unify it with a given premise. We terminate if no new facts have been generated.

Iterate over rules In this alternative implementation, we keep the two sets of facts $\mathcal{F}$ and $\mathcal{F}_n$ and iterate over the set $\mathcal{R}$ of rules. Initially, all facts generated during the pre-compilation phase are in the set $\mathcal{F}_n$ and $\mathcal{F}$ is empty. Given a rule with the premises $P_1, \ldots, P_n$, we try to find a fact f from the set $\mathcal{F}_n$ which unifies with P_1 up to P_n. If we succeed in unifying with P_i, we continue to search over the set $\mathcal{F}$ and $\mathcal{F}_n$ to find instantiations of the remaining premises. If all the premises are unifiable with some fact f, we generate a new fact P which is temporarily added to a set $\mathcal{F}'$, if there is no fact f' in $\mathcal{F}$, $\mathcal{F}_n$ or $\mathcal{F}'$ s.t. P is an instance of f'. This stage will terminate if all rules have been tried with the facts from $\mathcal{F}_n$. Now we add $\mathcal{F}_n$ to the set of facts $\mathcal{F}$ and $\mathcal{F}'$ will be used as our new set of facts $\mathcal{F}_n$. In this loop, the size of $\mathcal{R}$ remains constant. On the other hand, we may unify multiple times a given premise P_i with a given fact from $\mathcal{F}$. We terminate if no new facts are generated, i.e. $\mathcal{F}'$ is empty.

6.2 Experimental results

So far we have completed a prototype for the Horn fragment of LF. In this section, we discuss our preliminary experience and compare the performance with the tabled logic programming engine. We will pay particular attention to the two different implementation strategies of the inverse method. To evaluate and understand the current limitations, we will concentrate here on two examples, the first one computes the Fibonacci numbers, and the second one parses propositional formulas. All experiments are done on a machine with the following specifications: 3.4GHz Intel Pentium, 4.0GB RAM. We are using SML of New Jersey 110.55 under the Linux distribution Gentoo 16.14.under Linux. Times are measured in seconds, and the ∞ indicates we terminated the process after 30min.

Fibonacci example Computing the Fibonacci numbers is an interesting example, because a depth-first search will yield an exponential algorithm. Memoization allows us to re-use the computation of previous subgoals, and we expect its performance to be linear. Similarly, forward search has the potential of re-using results, and should yield a linear time algorithm. We compare the two different implementations for the inverse method, and the tabled logic programming engine.

			IR	IF	Tab	
k	fib(k)	Facts	Time	Time	Time	♯Entries
14	377	377(add) + 14(fib)	1.48	2.75	0.46 (0.08)	403
15	610	610(add) + 15(fib)	4.41	102.37	1.210 (0.07)	638
16	987	987(add) + 16(fib)	11.19	∞	3.135 (0.10)	1017
17	1597	1597(add) + 17(fib)	34.10	∞	6.861 (0.10)	1629
18	2584	2584(add) + 18(fib)	193.79	∞	139.826 (0.16)	2618

IF describes the inverse method where we iterate over facts and generate pre-instantiated rules, and IR denotes the inverse method where we iterate over the rules and the number of rules remains constant. The column Tab lists the runtime when all predicates are tabled. In parenthesis, we list the time if we selectively table only the fib predicate. The number of rules generated by the IF loop is 1470 for $k = 14$ and 2302 for $k = 15$. This is a staggering number compared to the 2 rules used in the IR loop. The high number of rules generated also yields a severe performance penalty. Tabling still outperforms inverse method search, even if we table all predicates in the program. As we can see, there is a severe penalty for tabling if we do not table selectively. In fact, selective tabling yields the best performance and does also outperform depth-first search.

Parsing Parsing algorithms are interesting since we typically would like to mix right and left recursive program clauses to model the right and left associativity properties of implications, conjunctions and disjunction. Clauses for conjunction and disjunction are left recursive, while the program clause for implication is right recursive. This program is not executable via depth-first search, and we compare the performance between the two implementations of the inverse method and tabling.

	IR		IF		Tab	
tokens	time	♯facts	time	♯fact	time	♯entries
5	0.860	138	0.109	2214	0.016	6
7	1.359	138	29.828	3702	0.015	10
9	1.016	138	33.391	3846	0.032	10
11	∞		∞		0.171	18

While the number of rules generated by the IF loop is not quite as large as for Fibonacci, it is still substantial. For 3 tokens, we generate 54 rules up to 182 rules for 9 tokens. This is compared to 13 rules which are generated during the pre-compilation phase in the IR method. These results clearly demonstrate that tabling cannot easily be outperformed. The inverse method is costly, and especially in the implementation IF the number of facts is growing substantially more. Our other

implementation of the inverse method where the number of rules remain constant has fair performance, although it cannot rival tabling.

To gain a better understanding of where the bottleneck lies in the inverse method implementation compared to a tabled implementation, we measured the number of unification failures. Unification is at the heart of proof search, and its performance affects in a crucial way the global efficiency of each of these applications. This is especially the case for the inverse method, since we rely on it to instantiate premises of rules, and to check for subsumption, i.e. is a newly derived uniform fact subsumed by an existing uniform fact. In the parsing example for example, we have over 3 million unification failures during subsumption checking, and over 21,000 unification failures when unifying a premise with a given fact. Let us contrast this to tabled logic programming where we count 70 unification failures all of which are in fact handled by the linear assignment algorithm. To check whether a new subgoal is already in the table no higher-order subsumption check is performed since we only check for α-variance. This strikingly illustrates that the performance of unification has a much greater impact on the inverse method than on tabled proof search.

7 Future Work and Conclusion

We presented the basis for an inverse method prover for the logical framework LF. Following standard development, we presented a forward uniform proof calculus and lifted it to allow for subformulas which may contain meta-variables. While we concentrate here on the logical framework LF, which is the basis of *Twelf*, it seems possible to apply the presented approach to λProlog [7] or Isabelle [10], which are based on hereditary Harrop formulas. Moreover, we proved the correctness of forward uniform proof calculus. Finally, we discuss challenges when implementing an inverse method prover for the logical framework LF.

In the future we intend to extend our implementation of the inverse method to hereditary Harrop formulas and cover the full higher-order fragment. To achieve a basic implementation seems not that difficult, however to build an inverse method prover with competitive performance we must tackle several issues. The first issue is efficient higher-order unification which seems central to the inverse method. Related to this issue is the fact that our theoretical development and implementation only deals with higher-order patterns where unification is decidable. To handle the full fragment of higher-order terms, we carefully need to revisit the issue of constraints.

Another important question is how to bound the inverse method search. While we do get a decision procedure when we execute the parsing algorithm with tabling, the inverse method does not directly yield a decision procedure. One way of addressing this problem may be to incorporate ideas from Chaudhuri *et al.* [3] and distinguish not only between left focusing and uniform proofs, but also introduce a right focusing phase. As observed in [3], this may have a substantial effect on performance. However, it remains unclear how to in general classify atoms as being left or right biased or mix the two biases. Extending the given theoretical framework to consider different bias for atoms is in principle possible.

Finally an important question is how to bring some goal-directed search into the inverse method. While the subformula property restricts the proof search on the level of formulas, it does not restrict the possible instantiations for the objects occurring in formulas. This has been already observed in the logic programming community and lead to the development of magic sets [16]. Magic sets transform the original program in such a way that a forward chaining logic programming engine is goal-directed and will only generate the relevant subgoals for a given query. Incorporating magic sets into the inverse method could substantially reduce the number of generated intermediate goals, and only generate relevant subgoals thereby yielding a competitive engine compared to backward chaining logic programming.

References

[1] B. Aydemir, A. Bohannon, M. Fairbairn, J. Foster, B. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, and S. Zdancewic. Mechanized metatheory for the masses: The poplmark challenge. In Joe Hurd and Thomas F. Melham, editors, *Proceedings of the Eighteenth International Conference on Theorem Proving in Higher Order Logics (TPHOLs), Oxford, UK, August 22-25*, volume 3603 of *Lecture Notes in Computer Science(LNCS)*, pages 50–65. Springer, 2005.

[2] Kaustuv Chaudhuri. The focused inverse method for linear logic. Technical report, Department of Computer Science,, December 2006. CMU-CS-06-162.

[3] Kaustuv Chaudhuri, Frank Pfenning, and Greg Price. A logical characterization of forward and backward chaining in the inverse method. In U. Furbach and N. Shankar, editors, *Proceedings of the Third International Joint Conference on Automated Reasoning, Seattle, USA*, Lecture Notes in Artificial Intelligence (LNAI). Springer-Verlag, 2006.

[4] Anatoli Degtyarev and Andrei Voronkov. The inverse method. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, pages 179–272. Elsevier and MIT Press, 2001.

[5] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.

[6] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

[7] Gopalan Nadathur and Dale Miller. An overview of λProlog. In Kenneth A. Bowen and Robert A. Kowalski, editors, *Fifth International Logic Programming Conference*, pages 810–827, Seattle, Washington, August 1988. MIT Press.

[8] Gopalan Nadathur and Dustin J. Mitchell. System description: Teyjus – a compiler and abstract machine based implementation of Lambda Prolog. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 287–291, Trento, Italy, July 1999. Springer-Verlag LNCS.

[9] Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. A contextual modal type theory. *ACM Transactions on Computational Logic (accepted, to appear in 2007)*, page 56 pages, 2006.

[10] Lawrence C. Paulson. Natural deduction as higher-order resolution. *Journal of Logic Programming*, 3:237–258, 1986.

[11] Brigitte Pientka. A proof-theoretic foundation for tabled higher-order logic programming. In P. Stuckey, editor, *18th International Conference on Logic Programming, Copenhagen, Denmark*, Lecture Notes in Computer Science (LNCS), 2401, pages 271 –286. Springer-Verlag, 2002.

[12] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[13] Brigitte Pientka. *Tabled higher-order logic programming*. PhD thesis, Department of Computer Sciences, Carnegie Mellon University, December 2003. CMU-CS-03-185.

[14] Brigitte Pientka. Tabling for higher-order logic programming. In Robert Nieuwenhuis, editor, *20th International Conference on Automated Deduction (CADE), Talinn, Estonia*, volume 3632 of *Lecture Notes in Computer Science*, pages 54–68. Springer, 2005.

[15] Frank Pfenning and Carsten Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 202–206, Trento, Italy, July 1999. Springer-Verlag Lecture Notes in Artificial Intelligence (LNAI) 1632.

[16] Raghu Ramakrishnan. Magic templates: a spellbinding approach to logic programs. *Journal of Logic Programming*, 11(3-4):189–216, 1991.